

Methods and parameters

● Introduction

Large programs can be complex, with the result that they can be difficult to understand and debug. The most significant technique for reducing complexity is to split a program into (relatively) isolated sections. This allows us to focus on an isolated section without the distractions of the complete program. Furthermore, if the section has a name, we can invoke it (cause it to be obeyed) merely by using this name. In a way, it enables us to think at a higher level. In Java, such sections are known as methods. We made extensive use of graphics methods to draw shapes on the screen in Chapter 3.

Recall the `drawRect` method, which we invoke with four parameters in this manner:

```
g.drawRect(20, 20, 80, 50);
```

First, the use of parameters allows us to control the size and position of the rectangle. This ensures that `drawRect` is flexible enough for a variety of circumstances. The parameters modify its actions. Incidentally, the correct Java terminology is *arguments*, but we shall use the more meaningful term *parameters*. Secondly, note that we *could* produce a rectangle by using four invocations of `drawLine` (in fact, if you were to look behind the scenes at the source code of `drawRect`, you would see that this is indeed how a rectangle is drawn). However, bundling up the four `drawLine` instructions inside a method known as `drawRect` is a sensible idea – it enables the graphics programmer to think at a higher level.

● Writing your own methods

Here, we will examine the construction of a method which draws an isosceles (two sides equal) triangle on a flat base, in the form of a tent. We are designing the method, so we can choose the parameters that will be required. For example, we could require three pairs of coordinates, but for our isosceles flat-based triangles, we will choose:

- the coordinates of the bottom left corner;
- the length of the base;
- the vertical height.

Rather than dive straight into writing a method, we will approach it in stages, introducing the basic statements involved in drawing a triangle, then moving towards bundling them up into a method with parameters.

Here are some Java statements which draw a triangle with a bottom corner 80 pixels in and 200 pixels down, with a base of 100 and a height of 110. First, we draw from the left end of the base to the right. Only the horizontal position changes:

```
g.drawLine(80, 200, 80+100, 200);
```

Secondly, we draw from the right of the base to the apex. We divide the base by 2 to find the mid-point.

```
g.drawLine(80+100, 200, 80+100/2, 200-110);
```

Thirdly, we draw from the apex to the bottom left.

```
g.drawLine(80+100/2, 200-110, 80, 200);
```

Rather than calculating each point as a single number, we have left some of them as expressions to show the calculations involved. Later we will simplify these calculations.

Altering the above triangle is difficult – but we could make it more flexible by introducing variables, as shown in this complete program:

```
import java.awt.*;
import java.applet.Applet;

public class TriangleTry extends Applet {
    public void paint(Graphics g) {
        int bottomX=80;
        int bottomY=200;
        int base=100;
        int height=110;

        g.drawLine(bottomX, bottomY, bottomX+base, bottomY);
        g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-
            height);
        g.drawLine(bottomX+base/2, bottomY-height, bottomX, bottomY);
    }
}
```

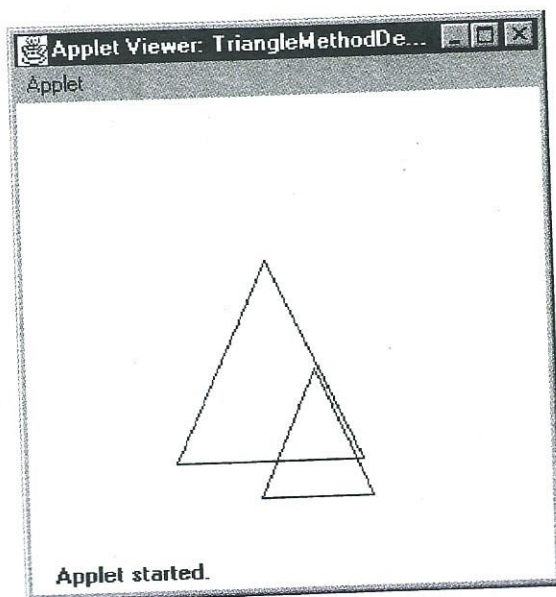
It is easy to change the triangle – simply alter the initial values of the variables. But what if we wanted to draw two triangles at different positions? We would have to duplicate the code! Instead we will create our own method, which we will call `drawTriangle`. (The choice of name is up to us.)

● A first method

Figure 5.1 shows a program containing a method named `drawTriangle`, which is invoked (made use of or called) twice. The resulting output is shown in Figure 5.2.

```
import java.awt.*;  
import java.applet.Applet;  
  
public class TriangleMethodDemo extends Applet {  
    public void paint(Graphics g) {  
        drawTriangle(g,80,200,100,110);  
        drawTriangle(g,125,220,60,70);  
    }  
  
    private void drawTriangle(Graphics g, int bottomX, int bottomY,  
                               int base, int height) {  
        g.drawLine(bottomX, bottomY, bottomX+base, bottomY);  
        g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-height);  
        g.drawLine(bottomX+base/2, bottomY-height, bottomX, bottomY);  
    }  
}
```

Figure 5.1

Figure 5.2 The output of the `TriangleMethodDemo` applet.

Yes, the program is short. However, the concept of methods and parameters is a major skill that all programmers need to master. We will now discuss the program in detail.

● Private and public

Look at the extract:

```
private void drawTriangle(Graphics g, int bottomX, int bottomY,
                        int base, int height)
```

This declares (introduces) the method, and is known as the *method header*. It states the name of the method (which we had the freedom to choose), and the parameters that it will accept. We shall examine parameters below. The rest of the method, enclosed in { }, is known as the *body*. Often the header consists of a long line, and we may choose to split it up at suitable points (though not in the middle of a word).

A vital decision that the programmer must make is: where can the method be invoked from? There are two main choices:

- The method can only be invoked from within the current applet or program. In this case, use the keyword `private`.
- The method can be invoked by another program. In this case, we would use the keyword `public`. (Methods like `drawLine` are examples of methods which have been declared as `public` – they are intended for general use.) Creating public methods involves a deeper knowledge of OO concepts, which we will leave till Chapter 9.

The private and public keywords are known as *access control modifiers*.

Moving on, we have the `void` keyword. This involves further choices:

- Will the method calculate a result and return it to the section of code which invoked it?
- Will the method perform a task without the need to supply an ‘answer’?

In the triangle method, no result is produced. (A picture on the screen does not count as a result – if we extended the method to calculate the area of the triangle as well, then *that* would be a result.) This possibility is covered later in the chapter.

● Invoking a method

In Java, you invoke a private method by stating its name, together with a list of parameters. In our program, the first invocation is:

```
drawTriangle(g, 80, 200, 100, 110);
```

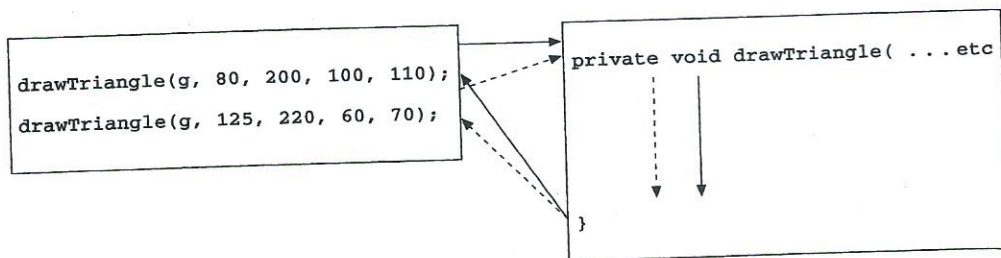


Figure 5.3 Invoking a method.

This statement has two effects:

- The parameter values are automatically transferred into the method. This transferring is a duplication process, in that the original values remain intact.
- The program skips to the body of the method – the part enclosed in { }, and executes the statements. When it runs out of statements and reaches the closing }, it continues its execution at the point where it was invoked from. Figure 5.1 shows paint invoking the same method twice.

The second invocation then takes place:

```
drawTriangle(g,125,220,60,70);
```

Figure 5.3 illustrates this. There are two invocations, producing two triangles.

● Passing parameters

It is essential to have an understanding of how parameters are transferred (passed) into methods. In our example, the concept is shown in the following lines:

```
drawTriangle(g,80,200,100,110);
void drawTriangle(Graphics g,int bottomX,int bottomY,int base,int height)
```

Recall our likening of a variable to a box. Inside the method, a set of empty boxes awaits the transfer of parameters. After the transfer, we have the situation shown in Figure 5.4. The transfer takes place in a left-to-right order. The invocation must provide the correct number and type of parameters. If the invoker (the user) accidentally gets parameters in the wrong order, the transfer process won't re-order them! When the drawTriangle method executes, the above values control the drawing process. Though we have invoked the method with numbers, we can use expressions (i.e. involving variables and calculations), as in:

```
drawTriangle(g,100,100,50+4,30);
drawTriangle(g,100,startY,60,tall);
```

In the above, the parameters are evaluated (reduced to a single value), then copied into the method. The technical term for this approach to passing parameters is *passing by*

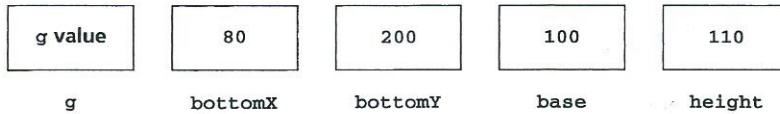


Figure 5.4 Passing parameters.

value, and this is how Java passes the built-in primitive types (`int`, `float`, `boolean` etc.). However, in Chapters 9 and 13 we will encounter a different approach known as *passing by reference*, which Java uses when objects (including arrays) are passed.

There is an additional parameter that we need to perform drawing. As we saw in Chapter 3, the applet viewer supplies us with a drawing area, which we named `g`. To enable other methods to use the same area, we need to pass it to them as a parameter.

● Formal and actual parameters

There are two lists of parameters that we are discussing, and it is important to be clear about the difference:

- The list of parameters that the invocation must supply is termed the *actual* parameters. Imagine this as meaning ‘the current parameters at this moment’.
- The list of names and types that the writer of the method decides on is termed the *formal* parameters. Though their values will change, the names stay the same. The writer of the method is free to choose formal parameter names. If similar names are used in other methods, no problem arises – each method has its own copy of its parameters.
- The type of each formal parameter must be chosen – this depends on the particular method. The type name must be placed before each parameter name, and a comma is used to separate the type/name pairs. Look at the `drawTriangle` header to see the arrangement.

SELF-TEST QUESTION

5.1 Explain what is wrong with these invocations:

```
drawTriangle(g, 100, 100, 50, "10");
drawTriangle(100, g, 100, 50, 10);
drawTriangle(g, 100, 100, 30);
```

● Local variables

Just as we declared variables in `paint`, so we might want to declare them within *any* method. The following program shows the use of variables to simplify the calculations in `drawTriangle`.

```

import java.awt.*;
import java.applet.Applet;
public class TriangleMethodDemo extends Applet {
    public void paint(Graphics g) {
        drawTriangle(g, 80, 200, 100, 110);
        drawTriangle(g, 125, 220, 60, 70);
    }
    private void drawTriangle(Graphics g, int bottomX, int bottomY,
                               int base, int height) {
        int rightX = bottomX + base;
        int topX = bottomX + base / 2;
        int topY = bottomY - height;

        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }
}

```

The variables `rightX`, `topX` and `topY` which we have introduced exist only within `drawTriangle`. If variables of the same name exist within other methods, then there is no conflict, in that each method uses its own copy. Another way to look at this is that when *other* programmers are creating methods they can invent local variables without cross-checking with everyone.

The role of local variables is to assist in the work of the method, whatever it is doing. The variables have a limited scope, restricted to their own method. Their existence is temporary – they are created when a method is invoked, and destroyed when it exits.

● Name clashes

In Java, the creator of a method is free to choose appropriate names for local variables and formal parameters – but what happens if names are chosen which clash with other variables? We could have:

```

private void methodOne(int x, int y) {
    int z = 0;
    // code...
}

private void methodTwo(int z, int x) {
    int w = 1;
    // code...
}

```

Let us assume that the methods have been written by two people. `methodOne` has `x` and `y` as parameters, and declares an integer `z`. These three items are all local to `methodOne`. In `methodTwo`, the programmer exercises the right of freedom to name local items, and opts for `z`, `x`, and `w`. The name clash of `x` (and of `z`) does not give a problem, as Java treats the `x` of `methodOne` as different from the `x` of `methodTwo`.

Let us summarize the method facilities we have discussed so far. Later we will include the `return` statement.

- The general form is:

```
private void someName(parameter list) {
    body
}
```

The programmer chooses the method name.

- The parameter list is a list of types and names, separated by commas. If a method doesn't need parameters, we use empty brackets for the actual and formal parameters, as in:

```
private void myMethod() {
    body
}
```

and the method invocation is:

```
myMethod();
```

- An applet can contain any number of methods, in any order. The layout is:

```
public class Any extends Applet {
    public void paint(Graphics g) {
        body
    }

    private void someName() {
        body
    }

    private void anotherName() {
        body
    }
}
```

● public and paint

There seems to be something different about `paint`. All our private methods are declared *and* invoked, but `paint` is only declared! There *is* something different. The applet viewer

(or browser) initiates your applet, and expects you to provide a declaration of a method called `paint`. The applet viewer then invokes `paint` as required. This also explains why `paint` cannot be private – it has to be available externally. When we look at event-driven programs in Chapter 6, we will see other similar public methods.

● return and results

When discussing parameters, we stressed that their values were copied into the formal parameters – this is a one-way process. If we need to get results *out* of a method, we must use a different mechanism – the `return` statement. Let us look at a simple method which calculates the area of a rectangle, given its two sides as parameters. We put:

```
import java.awt.*;
import java.applet.Applet;
public class ReturnDemo extends Applet {
    public void paint(Graphics g) {
        int answer = areaRectangle(30, 40);
        g.drawString("area of rectangle is "+answer, 100, 100);
    }

    private int areaRectangle(int side1,int side2) {
        int area = side1 * side2;
        return area;
    }
}
```

There are a number of new features in this example, which go hand in hand.

Instead of `void`, we have used a type – in this case `int`. This specifies that the method will use a `return` statement to pass back a value of that type. The choice of the type depends on the problem, but it can be `int`, `float`, a string or even a class type which you will encounter later: `Button`, `TextArea`, `Scrollbar` etc.

To return a value, we put:

```
return expression;
```

The expression (as usual) could be a number, a variable or a calculation (or even a method invocation), but it must be of the correct type, as specified in the declaration of the method – i.e. its header. Additionally, the `return` statement causes the current method to stop executing, and returns immediately to where it left off in the invoking method.

A method which returns a value cannot be used as a complete statement, as in:

```
areaRectangle(10,20);    //    wrong
```

Instead, the invoker must arrange to ‘consume’ the returned value.

Here is an approach to understanding the returning of values: imagine that the method invocation (the name and parameter list) is erased, and is replaced by the returned result. If the resulting code makes sense, then Java will allow you to make such an invocation. Look at this example:

```
answer = areaRectangle(30,40) ;
```

The result is 1200, which we imagine as replacing the invocation, effectively giving:

```
answer = 1200;
```

This is valid Java. But if we put:

```
areaRectangle(30,40);
```

the substitution would produce:

```
1200;
```

which is meaningless. Here are some more ways that we might consume the result:

```
int n;
n = areaRectangle(10, 20);
g.drawString("area is " + areaRectangle(10,20), 100, 100);
n = areaRectangle(10, 20) + areaRectangle( 22, 33);
```

SELF-TEST QUESTION

5.2 Work through the above statements with pencil and paper, substituting results for invocations.

To complete the discussion of **return**, note that it can be used with **void** methods. In this case, we must use **return** without specifying a result, as in:

```
private void demo() {
    //do something
    return;
    //do something else
}
```

This can be used when we want the method to terminate at a statement other than the last one.

● The flexibility of methods

Let us re-code our example:

```
import java.awt.*;
import java.applet.Applet;
public class ReturnDemo extends Applet {
    public void paint(Graphics g) {
        g.drawString(
            "area of rectangle is"+areaRectangle(30,40), 100, 100);
    }
    private int areaRectangle(int side1, int side2) {
        return side1*side2;
    }
}
```

Because we can use `return` with expressions, we have omitted the variable `area` in `areaRectangle`. Also, because expressions can be used as actual parameters, we have omitted the variable `answer` in `paint`; instead, we have invoked `areaRectangle` from a parameter of `drawString`. However, if we needed to use the same result lower down in `paint`, the use of a variable to memorize the number is more efficient than invoking `areaRectangle` twice.

Such reductions in program size are not always beneficial, because the reduction in meaningful names can reduce clarity, hence leading to more debugging and testing time.

● Building on methods

As an example of methods using other methods, let us create a method which draws a simple house cross-section of the form shown in Figure 5.5. We will choose the parameters to be:

- the bottom left coordinates;
- the width;
- the height of the walls.

All our houses will have similar proportions – the roof height will be half the wall height.

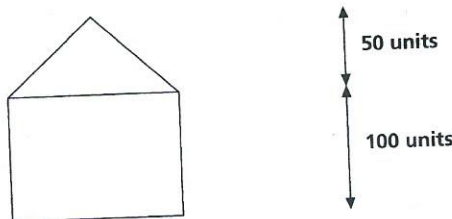


Figure 5.5 A simple house cross-section.

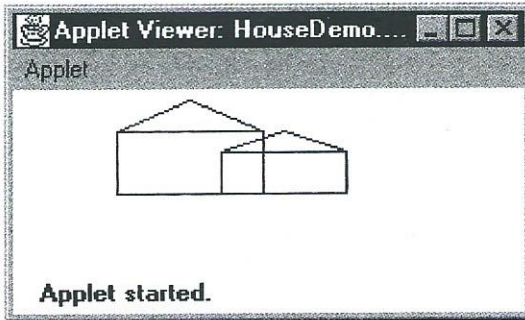


Figure 5.6 The output of the HouseDemo applet.

We will use `drawRect` from the Java library, and use our own `drawTriangle`.

This example illustrates the need for the `import` statement: the `drawRect` method is in fact contained in a prewritten library supplied with every Java system. If we omitted the statement:

```
import java.awt.*;
```

then `drawRect` would not be linked to our program. On the other hand, we don't need an include for `drawTriangle`, because it exists within our program.

Here is the program, with the resulting images shown in Figure 5.6.

```
import java.awt.*;
import java.applet.Applet;
public class HouseDemo extends Applet {
    public void paint(Graphics g) {
        drawHouse(g,50,50, 70,30);
        drawHouse(g,100,50,60,20);
    }

    private void drawTriangle(Graphics g,int bottomX, int bottomY,
                               int base,int height) {
        g.drawLine(bottomX, bottomY, bottomX+base, bottomY);
        g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-height);
        g.drawLine(bottomX+base/2, bottomY-height, bottomX, bottomY);
    }

    private void drawHouse(Graphics g,int bottomX,int bottomY,
                            int width,int height) {
        g.drawRect(bottomX, bottomY-height, width, height);
        drawTriangle(g, bottomX, bottomY-height, width, height/2);
    }
}
```

The program is straightforward if you recall that:

- The details about the drawing area (g above) need to be passed to any method that draws.
- Methods return to where they were invoked from, so:
 - paint invokes drawHouse;
 - drawHouse invokes drawRect;
 - drawHouse invokes drawTriangle;
 - drawTriangle invokes drawLine (three times).
- Actual parameters can be expressions, so height/2 is evaluated, then passed into drawTriangle.
- The bottomX and bottomY of drawHouse and the bottomX and bottomY of drawTriangle are totally separate. Their values are stored in different places.

You will see that what might have been a long and complex program has been written as a short program, split into methods with meaningful names. This illustrates the power of using methods.

Grammar spot

- The general pattern for methods takes two forms. First, when the method does not return a result, we declare the method by:

```
private void methodName(formal parameters) {
    body
}
```

and we invoke the method by a statement, as in:

```
methodName(actual parameters);
```

- When the method returns a result, the form is:

```
private int methodName(parameter list) {
    body
}
```

Any type or class can be specified, not only int.

- We invoke the method as part of an expression, e.g.:

```
int n = methodName(a, b);
```

The body of the method must include a return statement featuring the correct type of value.

- When a method has no parameters, we use empty brackets () in both the declaration and the invocation.

- The formal parameter list is created by the writer of the method. Each parameter consists of a type name (e.g. `int`, `float`), followed by a name. Commas separate the parameters.
- The actual parameter list is written by the invoker of the method. It consists of a series of expressions in the correct (matching) order, and of the correct types. Unlike formal parameters, the type names are not used.
- The appropriate `import` is needed when we use methods from library classes.

Programming pitfalls

- The method header must include type names. The following is wrong:

```
private void methodOne(x, f) // wrong
```

Instead we must put, for example:

```
private void methodOne(int x, float f)
```

- A method invocation must not include type names. For example, rather than:

```
methodOne(int y, float b);
```

we put:

```
methodOne(y, b);
```

- When invoking a method, you must supply the correct number of parameters and the correct types of parameters.
- If a method returns a value, you must arrange to consume its result in some way. The following style of invocation does *not* consume a return value:

```
someMethod(e, f);
```

New language elements

- The declaration of a private method:

```
private void someMethod(parameters) {
    body
}
```

- The use of `void` to indicate that a method does not return a result.
- The use of a type instead of `void` to state the type of a returned result.
- The invocation of a method, consisting of the method name and parameters.
- The use of `return` to simultaneously exit and pass a value back from a non-`void` method.
- The use of `return` to exit from a `void` method.

Summary

- Methods contain subtasks of a program.
- We can pass parameters into methods.
- Methods can return a result.
- A class can contain several methods.

EXERCISES

- 5.1** Code a method which draws a circle, given the coordinates of the centre and the radius. Its header should be:

```
private void circle(Graphics g, int xCentre, int yCentre,
                    int radius)
```

- 5.2** Code a method which draws a street of houses, using the provided `drawHouse` method. For the purposes of this question, a street consists of four houses, and there should be a 10 pixel gap between each house. The header is:

```
private void drawStreet(Graphics g, int wallHeight, int bottomX,
                        int bottomY)
```

where we provide the height of a wall, and the position of the bottom of the leftmost wall. The width of each house should be the same as the height.

- 5.3** Code a method (to be known as `drawStreetInPerspective`), which has the same parameters as Exercise 5.2. However, each house is to be 20% smaller than the house to its left.

- 5.4** Code a method `drawPerson` which draws a stick figure, formed from a circle for the head and lines for the torso, arms and legs. Base the size of the body parts on the height. The method header should be:

```
private void drawPerson(Graphics g, int height, int baseX,
                        int baseY)
```

where caller provides the height and the coordinates of the point between the feet.

- 5.5** Create a method `drawFamily` which draws two adults and two children (assumed to be half as tall as the adults). Make use of your `drawPerson`, and decide on your own parameters.

- 5.6** Write a method which returns the inch equivalent of its centimetre parameter. An example invocation is:

```
float inches = inchEquivalent(2.5f);
```

Multiply centimetres by 0.394 to calculate inches.

- 5.7** Write a method which returns the volume of a cube, given the length of one side.
A sample invocation is:

```
float vol = cubeVolume(1.2f);
```

- 5.8** Write a method which returns the area of a circle, given its radius as a parameter.
A sample invocation is:

```
a = area(1.25f);
```

The area of a circle is given by the formula $\pi \cdot r \cdot r$. An accurate value of π can be obtained by using `Math.PI`, which is available in a library. See Q 4.2 for details.

- 5.9** Write a method which returns the doubled value of its integer parameter.
A sample invocation is:

```
int d = doubled(n);
```

Explain how the following Java code is executed, and check your answer by running the program:

```
int d = doubled(doubled(doubled(10)));
```

ANSWERS TO SELF-TEST QUESTIONS

5.1 `drawTriangle(g, 100, 100, 50, "10");`

"10" should not be in quotes

`drawTriangle(100, g, 100, 50, 10);`

g should be the first parameter

`drawTriangle(g, 100, 100, 30);`

the number of parameters is incorrect

5.2 `int n;`

`n = areaRectangle(10, 20);`

n becomes 200

`g.drawString("area is "+areaRectangle(10, 20), 100, 100);`

The display is: area is 200

`n = areaRectangle(10, 20) + areaRectangle(22, 33);`

n becomes 200 + 726, i.e. 926